Appendix A: Greedy Alg rithm

This Appendix presents a formalized optimization method based upon the Greedy Algorithm. This method introduces the concept of local palette pattern for each texel. Therefore, it is not limited to any particular scheme of color distribution, and may be used for global palette construction in the case where each individual texel is indexed by the corresponding subset of palette entries, which is referred to as local palette.

The significant property of this approach is the pre-defined maximum number of iterations, which equals to the number of color values that are involved in at least one local palette. This strategy is more efficient than other methods in terms of time consumption.

The method constructs the global palette, which is understood as a depository for the entries of local palettes, only from those colors that are actually present in the original image (texture). This property may be important in some cases considering certain aspects of human vision system. The method can be modified so to remove this requirement, and Appendix C presents a method that does not impose this requirement.

Many variations on the general methodology presented herein are possible. For example, in some embodiments, color values associated with some texture blocks may be excluded from variation during optimization. This may be the case when a texels of a texture block or, group of texture blocks, have a single color value.

The method is not limited to any particular color representation, so any color model can be considered. Preferrably, the color space has a perceptually uniform or nearly perceptually uniform distance function. In practice, Euclidean metric with different weighs for RGB colors can be used as the adequate approximations of uniform color space.

The following notation will be used to develop the approach. Considering a finite color space Ω, $\rho(\cdot,\cdot)$ denotes the distance function, which is assumed to be nearly perceptually uniform. $X = \{x_i \in \Omega,\ i=1...N\}$ denotes the original texture of N elements. $Y = \{y_v \in \Omega,\ v=1...K\}$ denotes the global palette (i.e. the set of colors that are involved

at least in one local palette). $R = \{r_{iv} \in \{0,1\},\ i=1...N,\ v=1...K\}$ is the set of rules that define which entries from the global palette Y are to be taken as the local color palette entries for a texel. Thus, for a texel x_i the local palette includes y_v if and only if $r_{iv} = 1$. The set of rules R may be dependent upon the local palette pattern and modifications may be required for certain local palette patterns. Equation (1) specifies that at least one entry of global palette is available for any texel.

$$\forall i \sum_{v=1...K} r_{iv} > 0. \qquad (1)$$

$M = \{m_{iv} \in \{0,1\},\ i=1...N,\ v=1...K\}$ is an assignment matrix. It has exactly one value of '1' in each row. Position of '1' specifies which color from a global color palette is currently used for representing the corresponding texel. The assignment matrix M satisfies Equation (2), which is:

$$\forall i \sum_{v=1...K} r_{iv} m_{iv} = 1. \qquad (2)$$

Thus, the formalized problem of texture compression is the following. Given an arbitrary image (texture) X and a set of rules R, the objective is to find the global palette Y and the corresponding assignment matrix M (complying with (2)), such that the error between the compressed image (3),

$$X^* = M \times Y, \qquad (3)$$

and the original image X is minimal (the symbol × denotes here a standard matrix multiplication).

Human perception of the color of each pixel depends on the color of neighboring pixels (in addition to the color of the perceived pixel). However, for the sake of simplicity we consider the simplest model of human vision that does not consider the contribution of neighbor pixels.

Thus, the overall error is defined by Equation (4),

$$E(X,X^*)=\sum_{i=1...N}\rho\left(x_i,\sum_{v=1...K}m_{iv}y_v\right) \qquad (4)$$

As was introduced previously, the proposed algorithm defines exactly one palette entry per iteration. Thus, assign a Boolean value to each entry specifying whether or not it has been set up. Denoting this values with $S = \{s_v \in \{0,1\},\ v=1...K\}$, we state that before the algorithm begins all s_v are set to 0, and in the end, they are all equal to 1.

For a given image X, and current state of palette Y with respect to S, the following key functions are introduced. $\varepsilon(i, v)$ defined by Equation (5) indicates the error in pixel x_i if it is represented by the palette entry y_v provided that y_v is set (s_v=1) and available as a local palette entry according to R. If y_v is not yet set up or not included in the local palette, $\varepsilon(i, v)$ is equal to the value exceeding diameter of color space (we consider finite color spaces).

$$\varepsilon(i,v)=\begin{cases}\rho(x_i,y_v), \text{ if } s_v r_{iv}=1\\ MAX\_VALUE, \text{ otherwise}\end{cases} \qquad (5)$$

$E = \{\varepsilon_i,\ i=1...N\}$ is the set of errors for the current state of the palette construction. Each error is defined by Equation (6).

$$\varepsilon_i = \min_v \varepsilon(i,v) \qquad (6)$$

The function $d\varepsilon(v,x)$ defined by Equation (7) indicates the overall error decreases if the palette entry y_v is set to the value of x and then considered as defined. For already defined palette entries $d\varepsilon(v,x)$ equals 0.

$$d\varepsilon(v,x)=(1-s_v)\cdot\sum_{i:r_{iv}=1}\chi(\rho(x_i,x)-\varepsilon_i), \qquad (7)$$

where $\chi(t) = \begin{cases} t, if\ t > 0 \\ 0, otherwise \end{cases}$

The general idea of the algorithm is to find on each step the palette entry, that will maximally decrease overall error E if set up. This entry is considered defined, and all others (only those that may have changes by this assignment) should be recalculated to reflect the changes. The general idea of the algorithm can be expressed by the following steps:

(1) For each global palette entry y_v (that is not yet set up) we have to do the following:

a. Examine those texels, whose local palettes include y_v; and

b. Choose the one texel that would provide maximal error decrease if placed in y_v. In other words, we pick up the most representative candidate for the palette entry. As a consequence of this step, only original colors are included in the global palette Y.

(2) From all candidates (one per entry) we select one that provides maximal decrease compared to candidates of other entries.

(3) Set up the selected entry and execute the loop.

This strategy is relatively simple and produces very good results on different types of textures. The efficiency may be greatly improved by storing once calculated texel errors ε_i and error decreases $d\varepsilon_v = \max_{i:\, riv=1} d\varepsilon(v,x_i)$ in memory, and using them any time they are required for computations. Because setting up one palette entry on each step influences only limited number of texels, a relatively small number of errors and error decreases actually change and require recalculation.

Appendix B: Implementation of Greedy Algorithm f r Nodal Scheme of CD

In this Appendix we consider the nodal scheme of color distribution, as shown in Figure 7, for which an image is decomposed into equal-sized blocks and all texels within each individual block have the same local palette pattern. For this scheme, the general Greedy Algorithm (Appendix A) may be implemented more efficiently. The algorithm is based on minimizing the average degradation of each texel (i.e., selecting node colors so that distances from each texel (in color metric space) to nearest available node is minimal). Although the algorithm does not produce the theoretical minimum error the practical results are quite sufficient.

As shown in the local palette pattern of Figure 7, the present invention uses the same color value for color palettes of several adjacent blocks. Thus, the optimization procedure must analyze all blocks sharing a node to choose the preferred color value for that node. The algorithm provides a constant number of iterations equal to the number of texture blocks, since it sets up exactly one node per iteration minimizing overall error as much as possible. Nodes are assigned color values that are actually present in the original texture. This property is not required, but proved to result in compressed images of good visual quality.

As stated, the distance calculated from one color value to another may be based upon colors in RGB format or one of the standards established by CIE (Commission International de l'Eclairage) format. Weights, such as the exemplary weights in Figure 11, may be used for distance calculations using RGB format. Absolute differences or sum of squared differences can be used. Alternatively, CIE color space may be used. In this case, texels should be converted beforehand and stored in the required format. Weighting factors are not required to calculate color value distances in CIE space. Because CIE provides a better distance function than RGB, based upon human vision considerations, this format is preferred in order to obtain better image quality. However, CIE slightly slows down compression, and in most cases the quality improvement is not significant.

The algorithm uses the following structures, which stand for individual texel and node, respectively:

```
struct TEXEL {
    COLOR Color;      // Can be RGB or CIE
    double Error;     // Error of representing it by current palette
};
struct NODE {
    COLOR Color;      // Can be RGB of L*u*v* (L*a*b*)
    double Priority;  // Node priority
    bool SetUp;       // true if this color has been set up
};
```

Initially, all nodes are marked as not set-up and all texel errors are set to the maximum possible. All of the adjacent texture blocks for a node are inspected. Next, a color value is determined that, if assigned to a node, would maximally reduce the overall texel error. This color value is then assigned to the node and priority is set to the aggregate error decrease.

The node priority indicates how an aggregate texel error would decrease if a corresponding color value is considered as a final color value for the palette. The algorithm should decrease the overall error as much as possible, so a node with maximum priority is determined and marked as set-up (considering not set-up nodes only). This color value is used in the local palette of adjacent texture blocks.

Texel errors in adjacent texture blocks need to be recalculated to incorporate the changed color values at a node. Since this can in return change the color values and priorities of other adjacent nodes, these errors should also be recalculated. This procedure of setting up a node with maximum priority, decreasing texel errors, and recalculating influenced nodes continues until all nodes are set up. Finally, when a proper color value has been assigned to every node, an index is given to each texel based upon a comparison of the uncompressed texel color value and the best matching color in the color palette for the texture block.

Texel Error Calculation

The error for each texel indicates how well the color values in the current color palette represent a texel. In the very beginning, no colors are set up in the nodes, so the error is set to a maximum (e.g., 256.0 for RGB).

```
Texel.Error = 256.0;
for (each Node from AdjacentNodes(Texel) ) {
   if ( Not Node.SetUp ) continue; // Skip not set-up nodes
   Distance = ColorDistance( Texel.Color, Node.Color );
   if ( Distance < Texel.Error ) Texel.Error = Distance;
}
```

Thus, minimal distances to the colors available in adjacent set-up blocks are stored as the texel error.

Node Priority and Color Calculation

Priority indicates how the average error would decrease if color values were set up for the corresponding node. It is calculated by the procedure shown in the following pseudocode:

```
Node.Priority = 0.0;            // Minimal value
for ( each Texel from AdjacentBlocks(Node) ) {
   ErrorDecrease = 0.0;         // Decrease accumulator
   for ( each Texel1 from AdjacentBlocks(Node) ) {
      Distance = ColorDistance( Texel.Color, Texel1.Color );
      if ( Distance < Texel1.Error )
        ErrorDecrease += Texel1.Error – Distance;
   }
   if ( ErrorDecrease > Node.Priority ) {
      Node.Color = Texel.Color;
      Node.Priority = ErrorDecrease;
   }
}
```

Thus, all colors available in adjacent blocks are considered. The color value providing the maximum error decrease is determined and placed in a node. Priority is set to error decrease.

5

Nodes set up algorithm

The algorithm sets up one node at a time, according to priority. This procedure is represented by the following pseudocode:

```
// Initialization
for ( all Texels ) Texel.Error = 256.0;
for ( all Nodes ) CalculatePriority( Node );
// main loop
while ( Not all Nodes are set up ) {
   Node = NodeWithMaxPriority();
   Node.SetUp = True; // Set up node with maximum priority
   for ( each Texel from AdjacentBlocks(Node) )
       CalculateError( Texel );
   for ( each Node1 that shares the same block with Node )
       if ( Not Node1.SetUp ) CalculatePriority( Node1 );
}
```

When all blocks are set up, local color palettes for each block are available, and colors can be substituted by indices. This completes the texture compression processing.

Texel Clustering

The speed of this algorithm can be further improved by preliminary clustering of the colors within each block. A simple algorithm is proposed for clustering, which produces sufficient results and works very fast considering that each time it needs to cluster sixteen points.

Denote a set of points in a metric space as X. The task is to find clusters not exceeding d in diameter and comprising all points from X. As few clusters as possible should be used. At each step, the algorithm finds the diameter $[xy]$ of X. Then, it forms a cluster around x with diameter d and removes the covered points from X. The same procedure is also applied to y unless distance between x and y is less then d. In practice, this procedure works relatively well and is fast, although it does not construct a minimal set of clusters

Thus, clusters can be treated in the same way as texels in the iterative compression algorithm pseudocode. The only difference would be storing the number of texels belonging to each cluster, since this information is required for proper

calculation of ε_i and $d\varepsilon_l$, which represents texel error and node priority. The use of preliminary clustering makes the iterative algorithm 2-5 times faster, because the number of arithmetic operations is significantly reduced.

Decompression Algorithm

This section presents an algorithm for extracting one texel from a texture compressed according to the present invention. Texture decompression involves extraction of the index for a reconstructing texel and retrieval of the corresponding palette entry, typically stored locally. If the whole block is to be reconstructed, the decoder may reconstruct the whole palette, the color values for which are supplied by neighbor blocks. Then, after palette reconstruction, the decoder can reconstruct each texel by using its index to the color palette.

Decompression of the data encoded by the described above technique can be implemented very efficiently. In fact, since color interpolation is not used, no arithmetic operations on colors are required. Thus, the proposed technique is simpler than many prior art approaches. However, attention must be given to memory management, since color data is not stored locally as much as with prior block decomposition approaches. A Pseudocode for decompression of textures is presented below. We assume here, that colors and block indices are stored as separate 2-dimentional arrays. However, interlaced storage or other storage means are also possible.

```
RGB565 GetTexel( int x, int y ) {
block_x=x/4; block_y=y/4;
index=GetBlockIndex(block_x,block_y);
index=ExtractTexelIndex(index, x%4,y%4);
if (index&1) block_x++; // shift left
if (index&2) block_y++; // shift down
return GetColor(block_x,block_y);
}
```

In the above pseudocode, `GetBlockIndex()` and `GetColor()` are functions that are used for retrieving the index of the block and required color from the corresponding arrays. Since decoding typically takes place during three-dimensional scene rendering, data obtained by calling these functions is likely to be used for decompression of the next texel. This allows for more efficient use of memory cache (either standard or specially designed), because previously loaded indices and colors may be quickly accessed while they are in the cache. `ExtractTexelIndex()` is a function that extracts a two-bit index of the corresponding texel from the bits representing the indexes for all of the texels in the texture block.

Appendix C: Optimizati n By Iterative Conditi nal Mode

ICM (Iterative Conditional Mode) is another approach that can be used to optimize the color values for texels. ICM is conceptually similar to K-means clustering, which iterates through color space converging to the minimum of error function $E(X,X^*)$, defined by (4). The described method assumes that Euclidean metric is used in color space. This assumption, however, does not introduce any essential limitation on the general approach.

In the notation of Appendix A, if the global palette Y is defined, the assignment matrix M should comply with Equation (8) to minimize overall error for a given palette.

$$m_{iv} = \begin{cases} 1, \textit{if } v = \arg\min_{k:r_{ik}=1} \|x_i - y_k\|^2 \\ 0, \textit{otherwise} \end{cases} \qquad (8)$$

On the other hand, if the assignment matrix M is fixed, palette entries may be obtained by Equation (9), which is simplified by noting that the first derivative of $E(X,X^*)$ with respect to y_v is zero. The objective is to find Y, so that E(X,X*) converges to a minimum. The evident solution is to differentiate E with respect to Y, and set the derivative to 0. This is how the formula below was obtained.

$$y_v = \frac{\sum_{i=1\ldots N} m_{iv} x_i}{\sum_{i=1\ldots N} m_{iv}} \qquad (9)$$

Thus, the ICM-like algorithm alternately finds the assignment matrix for the fixed palette, and calculates palette colors as the median of the set of pixels that are indexed by this palette entry. This strategy is expressed by the following pseudocode:

```
M,Y CompressTexture( X,R ) {
    set y_v (v=1...K) to arbitrary colors;
    do {
        set m_iv (i=1...N, v=1...K) by (8);
        set y_v (v=1...K) by (9);
    } until (converged);
    return M,Y;
}
```

The ICM algorithm is quite efficient in most cases. However, it is known that this method frequently gets stuck at local minima, which should be carefully analyzed to get adequate performance. One strategy that may be applied in order to eliminate this problem may be found in *On Spatial Quantization of color images*, J. Ketterer at al., in Proc. of the European Conference on Computer Vision, 1998. Also, the ICM strategy can be accelerated by applying multiscale optimization techniques. For further information refer to *Multiscale minimization of global energy functions in some visual recovery problems*, Heitz et al., CVGIP: Image Understanding, 59:1, 1994.